Døñ’t Tòůčḥ Mý Dïąçr̥ıtīc͡s

Anonymous ACL submission

Abstract

The common practice of preprocessing text before feeding it into NLP models introduces many decision points which have unintended consequences on model performance. In this opinion piece, we focus on the handling of *diacritics* in texts originating in many languages and scripts. We demonstrate, through several case studies, the adverse effects of inconsistent encoding of diacritized characters and of removing diacritics altogether. We call on the community to adopt simple but necessary steps across all models and toolkits in order to improve handling of diacritized text and, by extension, increase equity in multilingual NLP.

1 Introduction

Virtually all natural language processing workflows begin with the ingestion of text data (with or without annotations). This text is not a random sample of written language, but usually has been sampled or filtered for language, script, quality, or relevance, and may also have been subject to various substitutions (e.g., case-folding, removing markup, or excising encoding errors). Yet preprocessing, as this process of text preparation is known, is generally regarded as more of a dark art than a topic for research, and as such has received minimal attention in the literature. While data preprocessing may not be the most important component of an speech or NLP system, any errors made at that stage are likely to propagate, and decisions made during preprocessing necessarily constrain what is possible downstream.

In this paper we focus our attention on the consequences of decisions made while preprocessing text with diacritics, a notion we define below. Using case studies, we show that failure to apply consistent Unicode normalization, which provides a canonical representation of text with diacritics, leads to degradation in downstream performance. We then show that stripping diacritics, common when pre-training neural network language models, also leads to degradation in various tasks.

Our recommendations, then, are simple: text preprocessing regimens should apply a consistent Unicode normalization—any of the normalization forms will do—but in most cases, should not attempt to strip diacritics.

2 Defining diacritics

Written text consists of atomic units sometimes called *glyphs*. These glyphs act as the primary spatial units in text and their order mirrors the temporal ordering of the orthographically relevant linguistic units in the corresponding utterances (Sproat, 2000). Glyphs may also bear non-spacing marks appearing above, below, to the left or right, or even surrounding, the glyph; it is these marks we will call *diacritics*. It is often difficult to discern whether marks of these sorts are really “part” of a glyph or glyphs on their own, and judgments also seem to vary from language to language, reflecting *Sprachgefühl* or conventions learned in school.[1]

Reflecting this ambiguity, Unicode often provides multiple ways to encode diacritized glyphs. For example, an *e* with an acute accent can either be encoded either as a single character ⟨é⟩ (`U+E9`), or as an *e* (`U+65`) followed by a combining acute accent (`U+301`). Similarly, consider the Hindi word साड़ी ‘sari’ (a traditional garment for women). In this word, one character is marked with a dot (a *nuqta*) underneath. With this dot the character is read as [ɽiː]; without it, it is read as [ɖiː]. In Unicode one can encode ⟨ड़⟩ either as a single precomposed character ड़ (`U+95C`) or as a sequence of the

[1] This uncertainty is not specific to diacritics. For instance, in *Gajica*, the Latin script used to write Serbo-Croatian, the digraphs ⟨dž⟩, ⟨lj⟩, and ⟨nj⟩ are considered single letters. Unicode, following earlier practices (in ISO 8859-2 and various vendor-specific encodings), provides unitary codepoints for these glyphs (and their uppercase and titlecase variants), even though they are often rendered the same as two-character sequences and they decompose into character sequences in compatibility normalization forms.

a.	बाढ़:	ब (U+92C)	ा (U+93E)	ढ़ (U+95D)			
	बाढ़:	ब (U+92C)	ा (U+93E)	ढ (U+922)	़ (U+93C)		
b.	عِدَّة:	ع (U+639)	ِ (U+650)	د (0x62F)	َ (0x64E)	ّ (U+651)	ة (U+629)
	عِدَّة:	ع (U+639)	ِ (U+650)	د (0x62F)	ّ (U+651)	َ (0x64E)	ة (U+629)

Table 1: Real-world Unicode canonicalization issues. (a): two different encodings of the Hindi word [bɑːɽʰ] 'flood', both found in the Hindi Dependency Treebank (Bhat et al., 2017; see §B). (b): two different encodings of the Arabic word [ʕiddah] 'number'; the former, in canonical ordering, is as it appears in the Prague Arabic Dependency Treebank (Smrz et al., 2008); the latter is the non-canonical order used in the Arabic Broadcast News Transcripts corpus (Maamouri et al., 2010).

undiacritized character (U+921) followed by a combining dot (U+93C).

It is straightforward to apply Unicode normalization (see Appendix A for a brief tutorial) to convert between these two representations, but without normalization ⟨é⟩ and ⟨é⟩, and ⟨ड़⟩ and ⟨ड़⟩, are considered unequal by ordinary string comparison methods (e.g., the C standard library function `strcmp`, or the == operator in Python) despite the fact they are visually indistinguishable. At the same time, some characters that might naïvely appear to be diacritized forms of others are not regarded as such by Unicode. For example, the "belted L" ⟨ł⟩ used in Polish (among other languages) does not decompose into ⟨l⟩ and a diacritic as one might expect, nor does the "O with stroke" ⟨ø⟩ used in various languages of Scandinavia decompose into ⟨o⟩ and a diacritic. One interesting comparison is between the "square script" used to write Modern Hebrew and the (Perso-)Arabic script used for Modern Standard Arabic. In the former, consonant pointing (e.g., *dagesh lene* and the *sin*/*shin* dot) is considered optional and Unicode regards the diacritic as a separate character. In the latter, consonant pointing (e.g., the dots distinguishing *sīn* and *shīn*, and *ṣād* and *ḍād* respectively) is mandatory and the points are treated as inalienable components of their glyphs.

Unicode also defines a canonical ordering for sequences of diacritics for those scripts in which a single glyph may bear multiple diacritics. For instance, in Arabic, in addition to the inherent consonant points, there are optional diacritics called *tashkīl*, including ones denoting quality of the following vowel (the *ḥarakāt*) or consonant gemination (*shaddah*). According to Unicode's canonical ordering, the *ḥarakāt* precede *shaddah*. Canonical ordering can be enforced by converting text to NFD or NFKD normalization forms.

3 The case for Unicode normalization

Applying Unicode normalization to text data enforces consistency with respsect to two dimensions. First, it ensures consistency in whether diacritics are precomposed or decomposed. Secondly, in scripts where a glyph may bear multiple diacritics, it ensures that these diacritic sequences are in a consistent order. The normalization algorithms are deterministic, conceptually simple, computationally efficient, and available in the standard library of nearly all modern programming languages.[2] Yet even some professionally developed corpora lack consistent normalization. Table 1 provides two real-world illustrations. Panel (a) shows two encodings (precomposed and decomposed) of a single word in the same corpus. Panel (b) shows two encodings (non-canonical and canonical diacritic ordering) of the same word in different corpora.

It is not difficult to show that the failure to apply a consistent (de)composition would have negative consequences for NLP systems. For example, the example in panel (a) of Table 1, drawn from Hindi Dependency Treebank, was part of the CoNLL 2017 shared task on dependency parsing. Using the un-normalized Hindi data, we first replicate the system of Straka and Straková (2017), using their UDPipe 1.0 model and published hyperparameters: we obtain an labeled attachment score (LAS) of 87.09 on the test set. However, simply by applying Unicode normalization form NFKC—which forces the *nuqta* characters to precomposed form—to the training and test data (and holding all other hyperparameters constant), we can improve the performance to a LAS of 87.38. While this improvement in LAS—0.29 absolute, 2.25 relative error reduction—may seem modest, it is available

[2] For C, C++, and Java, one is recommended to use the Unicode Consortium's open-source ICU library available at https://icu.unicode.org/.

more or less for free: normalization brings a number of visually-identical distinct words into equivalence.[3] Similar issues arise with the Arabic data in panel (b) in Table 1, when one attempts to use data from the two data sources in a single system without first applying normalization.

4 The case for preserving diacritics

Many preprocessing regimens have no need to canonicalize diacritics for the simple reason that they remove them altogether. Diacritics recognized as such by Unicode can be stripped from text by converting to either of the decomposition normalization forms (NFD or NFKD), which causes the diacritics to be treated as separate characters, and then removing all characters in Unicode's "Mark, nonspacing" (`Mn`) category. This procedure is, for instance, used by the normalizers in Hugging Face's `tokenizers` library. Others may also choose to remove characters in the "Control" (`C`) or "Separator" (`S`) categories. Náplava et al. (2018) devise a more aggressive method for stripping pseudo-diacritics (like the Polish belted L) using the Unicode full character names. For instance, the lower-case belted L ("Latin small letter L with stroke") is mapped onto lower-case L ("Latin small letter L") because the latter's name is a proper prefix of the former.

It is fairly obvious why one might choose to remove diacritics in this way. First, if diacritics are inconsistently encoded, it might be better to simply remove them. However, as just discussed above, it is trivial to apply a consistent encoding to text. Secondly, there are many scripts which are only rarely written with diacritics. In Arabic and Hebrew, *tashkīl* and *niqqud* diacritics, respectively, are omitted except in certain pedagogical and religious materials. The same is true of the diaeresis used in Russian text to distinguish ⟨ё⟩ [jo] from ⟨е⟩ [je], or the acute accent used to indicate stress. In many other scripts, diacritics are ordinarily present but occasionally omitted due to haste or technical challenges. For example, the contrast between ⟨l, ł⟩ is of some importance in Polish, but it is easy to imagine a Polish speaker typing ⟨l⟩ in place of ⟨ł⟩ because of limitations in the available text entry system.[4] In the presence of inconsistent diacritization, it might make sense to strip diacritics and thus use undiacriticized word forms as equivalence classes. While there are likely some applications where this is a sensible decision, in many others it comes with measurable costs. Our colleagues in the social sciences (p.c.) report to us that off-the-shelf Hebrew speech recognition systems output undiacritized text. When they attempt to feed the resulting text into NLP analysis pipelines (e.g., POS taggers and dependency parsers), even state-of-the-art systems struggle with the ambiguity introduced by the absence of diacritics and often produce incorrect outputs that could be avoided if the recognizer produced diacritized text and the NLP pipelines accepted it. Modern Greek, written using the modern *monotonic* conventions, makes consistent use of acute accents to mark primary stress. Without these accents, ambiguities arise; e.g., νόμος 'law, ordinance' vs. νομός 'county, district'. Yet GreekBERT (Koutsikakis et al., 2020), a pre-trained language model for Greek, is trained on text stripped of all diacritics.[5] Perhaps as a result, the model performs poorly when used for morphological analysis tasks (Yakubov, 2024). Kirov et al. (2024) study transliteration in twelve languages of South Asia. They experiment with a number of pre-trained language models, and find that one of them, mT5 (Xue et al., 2021), has poor vocabulary coverage in certain languages. This is because Malayalam and Telugu use the zero-width non-joiner character (`U+200C`) and Marathi and Sinhala use the zero-width joiner (`U+200D`), both to block inappropriate formation of conjunct (i.e., consonant cluster) characters, but these control characters are completely absent from the mT5 vocabulary. Presumably they were removed from the training data as part of an overzealous preprocessing regimen.

Idiosyncratic appearance of diacritics in both training data and inference can also have unexpected effects. In a preliminary survey of production machine translation systems for Hebrew, we found that MarianNMT (Junczys-Dowmunt et al., 2018) performs inconsistently whether a word in a source sentence in Russian contains an accent mark or not. In some cases, such as па́дают 's/he falls', the presence of the accent mark in the input produces incorrect translation ('grow') but the translation is correct when unaccented. A sim-

[3] Full details of this experiment are given in Appendix B.

[4] Older readers may have experienced a similar issues composing a text in their native language on a "dumbphone", or writing emails in their native language on a desktop computer while abroad.

[5] According to the documentation, the original "uncased" BERT checkpoints (Devlin et al., 2019) were also trained on stripped text.

ilar verb, опáли 'they fell' produces the opposite effect: the incorrect translation occurs when the word is unaccented. Source sentences in Spanish introduced a different phenomenon: in a sentence with a feminine-marked subject, an unaccented (and incorrect) form of the past participle *burlándose* 'mocked' is translated as masculine in Hebrew. The accented form is translated correctly. Since MarianNMT's training sets retain Spanish diacritics, we hypothesize unknown wordforms prevent the model from tracking grammatical agreement with nearby words; see Gonen et al. 2022 for similar observations.

Diacritization of undiacritized text, particularly in consonantal scripts like those used for Arabic and Hebrew, is now a well-established task in NLP which might help mitigate these effects in a manner similar to encoding canonicalization. However, this is something of an unsolved problem, as systems achieve just over 10% word error rate (WER) in Hebrew (Gershuni and Pinter, 2022). Similar WERs are reported for Arabic;[6] Náplava et al. (2018) report WER exceeding 40% for Vietnamese. It may also seem that pre-trained neural language models would be quite effective of this task—in a zero-shot, few-shot, or fine-tuning scenario—even if they have been trained on stripped text. While this certainly has been attempted, many of the state-of-the-art diacritization systems instead use randomly-initialized (rather than pre-trained) neural models. These models are robust, but are outmoded in most other NLP tasks: some may be surprised to find that a large amount of undiacritized text is less useful than relatively small amounts of in-domain diacritized data.

5 Conclusion

In this opinion paper, we argue and provide evidence that decisions about how diacritics are treated during text preprocessing may have detrimental downstream effects on model performance. These effects can largely be mitigated by preserving diacritics and by using simple, deterministic methods for ensuring they are encoded consistently. We believe the examples we have reviewed only scratch the surface. Many other issues may reside unseen deep within large, black-box neural models where they are difficult to detect.

Mitigation of these effects, while not particularly hard to implement, does however require the attention and effort of many stakeholders. Let us consider the case of large pre-trained models, used either for feature extraction (i.e., as pre-trained encoders) or prompting. Given a pre-trained model, even one with detailed documentation (e.g., model cards) it is often difficult to determine what text normalization steps were used to prepare data for the model. Furthermore, given such a model, it is not obvious how one might modify an existing checkpoint—or its tokenizer and vocabulary—of an pre-trained model to improve support for diacriticized text it has not yet been exposed to. Thus we are beholden to developers at institutions with sufficient resources to train such models and are in some sense stuck with the preprocessing decisions they make.[7] It is for this reason that we appeal to the community at large, rather than quietly modeling what we have argued to be best practices. Developers of large pre-trained models can increase their overall utilty, particularly for languages other than English, simply by applying a consistent normalization and resisting the urge to strip diacritics.

We note that this preprocessing issue is orthogonal to the ongoing debate concering the appropriate representation levels in language models. Byte-level and character-level models (e.g., Clark et al., 2022; Xue et al., 2022) are not immune to inconsistent encodings of diacritics, though it may be that they are better suited to represent inconsistent encoding than models with larger tokens. Vision-transformer processing of rendered text may help address invisible or near-invisible differences in rendered text (e.g., Lotz et al., 2023), but even here caution must be taken regarding any preprocessing performed before the rendering phase.

One might have expected that contemporary pre-trained neural models, given their impressive performance on a variety of tasks, would be robust to the use of diacritized inputs—or use in diacritization tasks—even when they themselves are not trained on diacritized text, but sadly this does not seem to be the case.

[6] It is difficult to cite any one WER as state of the art since there are many different diacritized Arabic corpora—some proprietary—used for evaluating diacritization, and error rates vary widely. Methodical system comparison across a variety of publicly-available corpora is desperately needed.

[7] For instance, Izsak et al. (2021), who describe how to train a BERT-style model on an "academic budget", do so with a cluster of GPUs that, at time of writing, would cost roughly $4,000 US on the second-hand market, not to mention other hardware and power costs. We suspect this budget is still well out of reach for most academic research groups.

Limitations

The case studies above reflect the authors' experience and issues reported in the literature. We suspect that many issues similar to those we discuss have been encountered by others but have either gone unnoticed or unpublished. By bringing attention to these issues, we hope to encourage researchers to not only pay greater attention to diacritics and other text encoding issues, but also to encourage them to discuss these and other preprocessing decisions in the literature.

References

Riyaz Ahmad Bhat, Rajesh Bhatt, Annahita Farudi, Prescott Klassen, Bhuvana Narasimhan, Martha Palmer, Owen Rambow, Dipti Misra Sharma, Ashwini Vaidya, Sri Ramagurumurthy Vishnu, and Fei Xia. 2017. The Hindi/Urdu Treebank Project. In James Pustejovsky and Nancy Ide, editors, *Handbook of Linguistic Annotation*, pages 659–698. Springer.

Jonathan H. Clark, Dan Garrette, Iulia Turc, and John Wieting. 2022. Canine: Pre-training an efficient tokenization-free encoder for language representation. *Transactions of the Association for Computational Linguistics*, 10:73–91.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Elazar Gershuni and Yuval Pinter. 2022. Restoring Hebrew diacritics without a dictionary. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 1010–1018, Seattle, United States. Association for Computational Linguistics.

Hila Gonen, Shauli Ravfogel, and Yoav Goldberg. 2022. Analyzing gender representation in multilingual models. In *Proceedings of the 7th Workshop on Representation Learning for NLP*, pages 67–77, Dublin, Ireland. Association for Computational Linguistics.

Peter Izsak, Moshe Berchansky, and Omer Levy. 2021. How to train BERT with an academic budget. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 10644–10652, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, et al. 2018. Marian: Fast neural machine translation in C++. *arXiv preprint arXiv:1804.00344*.

Christo Kirov, Cibu Johny, Anna Katanova, Alexander Gutkin, and Brian Roark. 2024. Context-aware transliteration of Romanized South Asian languages. *Computational Linguistics*, 50(2):475–534.

John Koutsikakis, Ilias Chalkidis, Prodromos Malakasiotis, and Ion Androutsopoulos. 2020. GREEK-BERT: The Greeks visiting Sesame Street. In *11th Hellenic Conference on Artificial Intelligence*, SETN 2020, page 110–117, New York, NY, USA. Association for Computing Machinery.

Jonas Lotz, Elizabeth Salesky, Phillip Rust, and Desmond Elliott. 2023. Text rendering strategies for pixel language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10155–10172, Singapore. Association for Computational Linguistics.

Mohamed Maamouri, Ann Bies, Seth Kulick, Wajdi Zaghouani, Dave Graff, and Mike Ciul. 2010. From speech to trees: Applying treebank annotation to Arabic broadcast news. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta. European Language Resources Association (ELRA).

Jakub Náplava, Milan Straka, Pavel Straňák, and Jan Hajič. 2018. Diacritics restoration using neural networks. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Otakar Smrz, Viktor Bielický, Iveta Kourilová, Jakub Krácmar, Jan Hajič, and Petr Zemánek. 2008. Prague Arabic Dependency Treebank: A word on the million words. In *Proceedings of the Workshop on Arabic and Local Languages (LREC 2008)*, pages 16–23.

Richard Sproat. 2000. *A Computational Theory of Writing Systems*. Oxford University Press.

Milan Straka and Jana Straková. 2017. Tokenizing, POS tagging, lemmatizing and parsing UD 2.0 with UDPipe. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 88–99, Vancouver, Canada. Association for Computational Linguistics.

Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. 2022. ByT5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306.

Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua,

and Colin Raffel. 2021. mT5: A massively multilingual pre-trained text-to-text transformer. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 483–498, Online. Association for Computational Linguistics.

Daniel Yakubov. 2024. How do we learn what we cannot say? Master's thesis, Graduate Center, City University of New York.

A Unicode normalization forms

Unicode provides four normalization forms, which are defined in terms of two types of equivalence:

- Codepoint sequences are said to be *canonically equivalent* if they have the same meaning and the same appearance when printed or displayed.
- Codepoint sequences are said to be *compatible* if they may have distinct appearances but the same meaning in certain contexts.

The normalization forms are computed as follows:

- `NFC`: decompose characters according to canonical equivalence, then recompose them according to canonical equivalence.
- `NFD`: decompose characters according to canonical equivalence, then order sequences of combining characters in canonical order.
- `NFKC`: decompose characters according to compatibility, then recompose them according to canonical equivalence.
- `NFKD`: decompose characters according to compatibility, then order sequences of combining characters in canonical order.

The Python standard library module `unicodedata` provides a function `normalize`, and this can be used to convert text strings between the four Unicode normalization forms. This function takes as its first argument the normalization form (e.g., `"NFD"`) and the string to be normalized as the second argument, returning the string in the desired normalization form.

B Hindi dependency parsing

Hindi experiments were were conducted using UDPipe v1.2.0 (https://github.com/ufal/udpipe, tag `v1.2.0`), word2vec (https://github.com/tmikolov/word2vec, commit `20c129a`), and the Hindi Dependency Treebank (https://github.com/UniversalDependencies/UD_Hindi-HDTB/, commit `54c4c0f`), targeting the "gold tokenization" subtask. Encoding inconsistencies with the Hindi treebank were reported by the authors to the maintainers, and this was marked fixed in commit `da32dec` (Dan Zeman, personal communication).